Automatic Speech Recognition 2012–13: Coursework lab (part 1 of 2)
— Continuous speech recognition —

Hiroshi Shimodaira and Steve Renals (*Revision* : 2.0)

In this lab session you will

- train monophone models on the WSJCAM0 database using HTK[1]
- investigate how recognition accuracy changes with more training
- try different amount of pruning during recognition (and possibly during training)
- try different values of the language model scaling
- see what effects that has on accuracy
- increase the number of Gaussian mixture components in the pdfs for the monophone models
- see what effects that has on accuracy
- convert monophone models into triphone (i.e. context dependent) models using a decision tree-based clustering technique.
- convert the single Gaussian triphone HMMs into multiple mixture component triphone HMMs.
- see what effects those have on accuracy

1 Preparation

1.1 Initialisation

You should use the following work directory ('*WorkDir*' here after) that is allocated to you in this course throughout the experiments.

`/afs/inf.ed.ac.uk/group/teaching/asr/Work/` *YourLoginName*

where *YourLoginName* denotes your login name.

```
% cd WorkDir
% /afs/inf.ed.ac.uk/group/teaching/asr/bin/asr-init.sh
```

You will find the following directories under *WorkDir*.

dir names	contents
corpus/	original corpus (wsjcam0) and parameterised data (WSJCAM0)
file_lists/	lists of files for training and recognition
labels/	phone and word labels
model_lists/	list of HMM model names
dictionaries/	word lists and pronunciation dictionaries
language_models/	language models
perl/	useful perl scripts called by some of the functions
configs/	HTK configuration files
edfiles/	HTK edit command files
logs/	log files
recognition/	recognition output
manuals/	copy of the HTK book, reference papers

You will also find the files `HTK_WSJCAM0.py` and `trainWSJCAM0.py`. `HTK_WSJCAM0.py` contains a number of python functions which call the relevant HTK programmes used for training and recognition. `trainWSJCAM0.py` contains the sequence of python function calls corresponding the procedure outlined in this document.

The online manual of HTK is available at this url:

WorkDir`/manuals/htkbook/index.html`

[1] `http://htk.eng.cam.ac.uk`

HINT If you want to avoid typing the long path name of *WorkDir*, it would be a good idea to create a symbolic link somewhere under your home directory. For example, if you've already had `~/asr` , the following will create a symbolic link, `~/asr/ASR`, whose path name can be used instead of *WorkDir*.

```
% cd ~/asr
% ln -s WorkDir ASR
```

1.2 Data

Data is from the WSJCAM0 Cambridge Read News corpus provided by LDC

`http://www.ldc.upenn.edu/Catalog/LDC95S24.html`

It is already converted to MFCCs, so need to do this yourself (and no need to experiment with different parameterisations) and those MFCC files can be found under

`corpora/WSJCAM0`

Do not copy the data either, use the central copy (the scripts are set up to do this).

There are about 8000 training utterances (from 90 speakers), 700 development utterances (from 20 speakers), and 20000 test ones (from 20 speakers).

The master copy of WSJCAM0, which includes speech wave files, can be found at under corpora/wsjcam0, which is a symbolic link to

`/group/corpora/public/wsjcam0`

The corpus is divided into sub sets for experiments as follows:

training set	si_tr
development sets	si_dt5a, si_dt5b, si_dt20a, si_dt20b
evaluation sets	si_et5a, si_et5b, si_et20a, si_et20b

where "5" and "20" denote a data set with vocabulary of about 5000 words and about 20000 words, respectively. We'll work with the 5000 word development set si_dt5a.

For details, please see those PDF files under the manual directory, `manuals`

"fransen-pye:wsjcam0:1994.pdf"	corpus description
"robinson-fransen:icassp:1995.pdf"	evaluation

2 Monophone models

A typical procedure for training phone HMMs with HTK is as follows, where typical HTK command names are shown in parentheses as well. As mentioned above, all the functions are in python, defined in `HTK_WSJCAM0.py`. Please refer to `trainWSJCAM0.py` as you go through this document.

- P1: with a small size of speech corpus in which phone labels are available, train each phone HMM individually. (HCompV → HInit → HRest)
- P2: with the same data set, carry out forward-backward training, referred to as "embedded training", for the set of HMMs. (HERest)
- P3: with a large size of speech corpus with sentence/word-level transcriptions, train the set of HMMs again. (HERest)

A typical sequence of procedures for an experiment will consist of following three steps:

	Descriptions	Python functions (HTK commands)
Step 0:	Preparation of speech data dictionary language model	
Step 1:	**Model training**	
(a)	model initialisation	`prepMonophones(hmmType=monoSingle, trainingID='hmmP', modelFile=initialMonoModelList)` (HCompV)
(b)	initial training	`initMonophones(hmmType=monoSingle, prepID='hmmP', initID='hmmI', restID='hmmR', modelFile=initialMonoModelList)` (HInit,HRest)
(c)		`mergeMonophones(hmmType=monoSingle, inID='hmmR', outID='hmm0', modelFile=initialMonoModelList)`
(d)	embedded training	`trainHMMs(hmmIter=monoSingleIter, n=nIters, hmmType=monoSingle, modelFile=initialMonoModelList, labelFile=initialPhoneTrainingMLF)` (HERest)
(e)	silence model refinement	`createSPModel(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList, edFile=silEdFile)`
(f)	label realignment	`alignMLF(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList, edFile=mergeSpSilEd, realignedMLF=realignedMonoMLF)` (HVite)
(g)	embedded training	`trainHMMs(hmmIter=monoSingleIter, n=nIters, hmmType=monoSingle, modelFile=spMonoModelList, labelFile=realignedMonoMLF)` (HERest)
Step 2:	**Recognition**	`recogniseSpeech(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList)` (HVite)
Step 3:	**Results analysis**	`scoreWER(hmmIter=monoSingleIter, hmmType=monoSingle)` (HResult)

When you the python functions you need to be in your *WorkDir*directory (and nowhere else); also you need to import the functions using `from HTK_WSJCAM0 import *`, and to set the variables defined at the top of `trainWSJCAM0.py`. The easiest way is probably to create your own version (e.g. `myTrainWSJCAM0.py`).

Step 0 has been done already. We shall go through the other steps in this section. There are some parameters you can change to investigate the effect to the performance of training or recognition.

For details on HTK commands, please refer to the HTK Book, whose copy is available in the manual directory.

2.1 Step 1: Model training

2.1.1 Model preparation

First of all, make sure that you are in the right directory, i.e. the top directory of this project (*WorkDir*).

```
% cd WorkDir
```

If you are using python interactively, remember to import the functions using `from HTK_WSJCAM0 import *`, and to set the variables as in lines 5–25 of `trainWSJCAM0.py`

Initial phone models (HMMs) have been stored in `proto/`, where you will find 45 files, each of which corresponds to a phone HMM.

Since HMMs are stored in ascii format at this stage, you can take a look at them. Try

```
% less proto/zh
```

you will see that models parameters such as mean vectors, variances of each state and transition probabilities have been set to default initial values. For details on HMM definition files, please refer to HTK Book section 7.2.

Now, call the following python function, which calls HCompV to initialise model parameters based on training data.

```
prepMonophones(hmmType=monoSingle, trainingID='hmmP', modelFile=initialMonoModelList)
```

The output is saved in directory `models/R1/hmmP`. Take a look at a file in the directory to see which parameters have been updated by the command. `R1` is the phase of training defined by `monoSingle` (i.e. monophones with single Gaussian distributions).

2.1.2 Initial training

Run this command:

```
initMonophones(hmmType=monoSingle, prepID='hmmP', initID='hmmI', restID='hmmR',
modelFile=initialMonoModelList)
```

This function runs HInit and HRest to train each phone model separately by referring phone labels given in a Mater Label File (MLF), `labels/phone/si_tr.mlf` . New models are saved in `models/R1/hmmI` (by HInit) and `models/R1/hmmR` (by HRest).

Detailed logs of the commands are saved in directory, `logs/init_monophones` .

2.1.3 Embedded training

So far, each phone HMM has been stored in a separate file. Hereafter, all the model files are saved into a single file for efficiency.

Run the following python function

```
mergeMonophones(hmmType=monoSingle, inID='hmmR', outID='hmm0',
modelFile=initialMonoModelList)
```

which will create a merged model file, `models/R1/hmm0/MODELS` .

Run the following function to carry out "*embedded training*" (HERest) iteratively

```
trainHMMs(hmmIter=0, n=4, hmmType=monoSingle, modelFile=initialMonoModelList,
labelFile=initialPhoneTrainingMLF)
```

where the `hmmIter` keeps count of the current iteration of HMM training and the argument `n=4` specified 4 iterations of forward-backward training. If This function first trains the models in `hmm0` and stores the output into `hmm1`, which is then used as an input for next training to produce `hmm2`, and so on. As a result, the newest model will be `hmm4`.

2.1.4 Silence model refinement

Run this function to create another silence model, *sp*, which is the short pause model which puts an optional pause between words.

```
createSPModel(hmmIter=4, hmmType=monoSingle, modelFile=spMonoModelList, edFile=silEdFile)
```

This will store a new model set in `hmm5`. For details, please see HTK Book section 3.2.2.

Retrain the models (which now include *sp*) using `trainHMMs`

```
trainHMMs(hmmIter=5, n=2, hmmType=monoSingle, modelFile=spMonoModelList,
labelFile=initialPhoneTrainingMLF)
```

which calls HERest twice, and the newest models will be now `hmm7` .

2.1.5 Label realignment

Now, it's time to create new phone labels using the current models, using this python function call:

```
alignMLF(hmmIter=7, hmmType=monoSingle, modelFile=spMonoModelList, edFile=mergeSpSilEd,
realignedMLF='labels/phone/mono-algned2.mlf')
```

which will crate a new MLF, `labels/phone/mono-algned2.mlf` (to which we have set the value of variable `realignedMonoMLF`).
(This will take a couple of minutes.)

2.1.6 Model retraining with new labels

Run the `trainHMMs` again to retrain the models.

```
trainHMMs(hmmIter=7, n=2, hmmType=monoSingle, modelFile=spMonoModelList,
labelFile=realignedMonoMLF)
```

In the above example, we repeat embedded training two times. However, the optimal number of iteration varies depending on the data you use. You could investigate how the number of training iteration would affect recognition accuracy on test data.

2.2 Step 2: Recognition

Assuming the newest model set you got is `hmm9`, run the following command

```
recogniseSpeech(hmmIter=9, hmmType=monoSingle, modelFile=spMonoModelList)
```

which runs HVite on test data set ("si_dt5a") using `hmm9` in `models/R1`.
The recognition output is stored in the directory, `recognition/R1` .

```
% ls -l recognition/R1                    ··· you will get a list of recognition output files
% less recognised/R1/hmm9_si_dt5a_output.mlf
```

Recognition speed and accuracy vary depending on the parameters.
Try turning pruning off altogether...

2.3 Step 3: Result analysis

Run the script to get results

```
scoreWER(hmmIter=9, hmmType=monoSingle)
```

Make sure you understand the output from `HResults` (HTK Book section 3.4.1 and 17.19.1).

BEWARE: If you have too much pruning, the decoder (HVite) will fail on some files and give an error message of "No token survived ...", and then `HResults` will not give you correct statistics because it reports results only on actually recognised output.

– so make sure it reports 368 sentences (for "si_dt5a" set), i.e. $N = 368$, or at least, not *too many* fewer than that, otherwise you might get misleading WER figures.

3 Experiments

3.1 things to investigate

1. Plot WER against at least those parameters:
   - pruning level
   - language model scaling factor
   - number of training iterations

   whose details are described in following sections.
2. Measure speed

3.2 Parameters for recognition

`HVite` has several parameters you might want to change. Plot graphs of WER against the value of each parameter (holding other parameters fixed). Note that the effects of the parameters interact.

It is probably best to write your own python scripts to automate this process, and collate the results automatically.

- **Pruning**: (`-t` flag to `HVite`, start with values around 90, say 50 to 200) Stick to the same HMMs (a fully trained set, not iteration 0). Alter the script to only use one of the four test sets if things are running slowly (but do not plot results for one test set on the same plot as for another, or all four). The scripts are provided with beam width set to 70, which probably is too harsh but makes the recogniser run fast.
- **Language model scaling factor**: (`-s` flag to `HVite`). When the log probabilities for acoustic model and language model are summed, the LM can be weighted. The scripts use a reasonable default, try varying the value either side of this. This parameter will need re-tuning when you try a different language model later on.
- **Word insertion probability**: (`-p` flag to `HVite`). Controls insertion/deletion ratio. A good rule of thumb is to make number of insertion and deletion errors equal. I would optimise this parameter after setting the LM scaling factor.

3.3 Parameters for training

During training, you can vary the amount of α and β pruning by changing the argument `-t 100 100 600`, where the first 100 is a log probability beam width. If training fails at that level of pruning (i.e. all αs and βs get pruned), HERest will add another 100 (the second number after the `-t`) and try that utterance again. It will keep adding 100 until it reaches 600, then give up. **Leave this until later**: you can return to this section and try tightening up that pruning to see if good models are still trained. For now, just train some models up using the current parameters. To understand what α and β pruning is, you need to understand the derivation of the Baum-Welch algorithm.

3.4 Measuring speed

You will have noticed that some scripts preface commands with `/usr/bin/time`. Unsurprisingly, this reports the time a program takes to run (try `time ls`), which would be shown in the last lines in the log file.

You should record the "user" time and **not** the "real" time. The latter is the actual elapsed time which will be longer than the time the process actually spent running on the CPU, since UNIX machines run many processes at once by allowing them to take turns on the CPU. On a heavily loaded machine, the "real" time may be considerably more than the "user" time. The "system" time is the time spent waiting for things like file access and should generally be small. Note that the format of the output from `time` may vary slightly across machines running different versions of Unix.

IMPORTANT: to make speed comparisons, always measure the speed on *the exact same type of machine*. Note that a couple of the machines in the lab (at the front) are slightly different hardware to the rest.

3.5 Hints

- Train a set of models and do several recognition experiments *which means you don't need to retrain the models each time*. Then retrain the models with different pruning, and do the same recognition experiments.
- Try and keep the machines busy - you can even do testing on one model set whilst training another, to save time, just don't get in a muddle. And, as ever, keep a clear record of your experiments and results.

4 Multiple mixture component monophone models

After you have carried out the embedded training of single Gaussian monophone models several times to get hmm9, it's time to increase the number of mixture components of each state. This is done by iterative "mixture splitting" with the HHEd command.

4.1 Copying source models to a new directory

At first, you need to determine the model number to increase the mixture component. If it is "`models/R1/hmm9`" [2], do as follows (here I assume your current directory is *WorkDir*):

```
copyModels(hmmIter=9, oldType=monoSingle, newType=monoGMM)
```

4.2 Mixing up

The following python function which calls HHEd, to split a single Gaussian to two components.

```
mixupGMM(hmmIter=0, hmmType=monoGMM, mixEdFile='edfiles/mixup2.hed',
modelFile=spMonoModelList)
```

Note that we have changed the model phase to `monoGMM`. This automatically creates a new directory, `models/R2/hmm1`, for new models.

You can take a look at the new models to see what changes have been made:

```
% less ./models/R2/hmm1/MODELS
```

4.3 Retraining

The new models above need retraining. Again we can use `trainHMMs` to call HERest (3 iterations in this example) creating hmm4.

```
trainHMMs(hmmIter=1, n=3, hmmType=monoGMM, modelFile=spMonoModelList,
labelFile=realignedMonoMLF)
```

4.4 Evaluation

Test the new models in `models/R2/hmm4` with the `recogniseSpeech` function, with `smallFlag` set to be true. uses a small test set, "`file_lists/si_dt5a-div3.scp`" (123 utterances) rather than the subset of "`file_lists/si_dt5a.scp`" (368 utterances). Using a small test-set can make the evaluation much faster, but it might lose accuracy in estimating recognition performance. However, we need to carry out a large number of experiments in the lab sessions, we will use this small subset for evaluation.

```
recogniseSpeech(hmmIter=monoGMMIter, hmmType=monoGMM, modelFile=spMonoModelList,
smallFlag=True)
```

which will produce recognition output `recognised/R2/hmm4_si_dt5a-div3_output.mlf` and log files in `logs/recognise/R2`. Make sure that you have not got the warning message of "No token survived..." many times (a few times should be fine).

As before use `scoreWER` to get recognition accuracy.

```
scoreWER(hmmIter=monoGMMIter, hmmType=monoGMM, smallFlag=True)
```

Are the new models more accurate than the single-Gaussian component models from earlier? (It would be interesting to evaluate other previous models, e.g. hmm1, hmm2, hmm3.)

4.5 Increasing more number of mixture components

We can repeat the above splitting and retraining operation to get models with more number of mixture components.

The question is how we should change the numbers of mixture components along with repetitive training. Recall the local-optimum problem with HMM parameter estimation - it would not be a good idea that we increase the number of mixture components very quickly, e.g. $1 \rightarrow 10 \rightarrow 50$. The following table shows an example of the process:

[2] Further experimental results will depend on which model you've chosen here. It would be interesting to see how different performance you will get depending on the initial model number.

# of mix. components	splitting	retraining	final model
(1)	-	-	(R1/hmm9)
2	hmm0 → hmm1	hmm1, hmm2, hmm3	hmm4
3	hmm4 → hmm10	hmm11, hmm12, hmm13	hmm14
4	hmm14 → hmm20	hmm21, hmm22, hmm23	hmm24
5	hmm24 → hmm30	hmm31, hmm32, hmm33	hmm34
7	hmm34 → hmm40	hmm41, hmm42, hmm43	hmm44
10	hmm44 → hmm50	hmm51, hmm52, hmm53	hmm54
⋮	⋮	⋮	⋮

Try increasing the number of mixture components further to see how recognition accuracy/speed changes, and draw graphs[3]. To that end, read the next section 4.6. There are several files in `edfiles` for mixing up to e.g. 4, 8, 16 components.

Go over your notes and make sure you understand what you just did, what mixtures of Gaussian are, and why increasing the number of mixture components improves accuracy.

4.6 Details on mixup and HHEd commands

The `mixupGMM` function calls HHEd in the following manner:

```
HHEd -C configs/config.basic -T 1 -H models/R2/hmm0/MODELS
-M models/R2/hmm1 edfile/mixup2.hed model_lists/mono-sp.list
```

Here `models/R2/hmm0/MODELS` is used as source models, and a new model file is stored in the directory `models/R2/hmm1`. Operation (edit) commands such as increasing the number of mixture components should be given to HHEd through a file, which is specified as the 1st argument to HHEd.

As you will see, the edit command file, "`edfile/mixup2.hed`", consists of only one line:

```
MU 2 {*.state[2-4].mix}
```

where "2" followed by "MU" is the number of Gaussian mixture components you want to have for each state (i.e. 2, 3, and 4) of all the models.

5 Decision tree-based tied-state triphone models

We can improve the monophone models by making them context dependent. Among several types of training algorithms for context-dependent models, we will employ a decision tree-based clustering technique to have triphone models whose parameters are effectively shared, i.e. "tied".

The procedure will look like this

Step 1a create triphone labels and triphone model list.

Step 1b create single-Gaussian cross-word triphones by cloning single-Gaussian monophone models.

Step 1c train single mixture cross-word triphones.

Step 2a carry out clustering to create single-Gaussian tied-state triphone models from the models above.

Step 2b train the tied-state triphone models.

Step 3 split Gaussian component to get multiple-mixture tied-state triphone models, and train and evaluate the models.

5.1 Step 1: Cross-word triphone models

5.1.1 Step 1a: preparing triphone labels

`createTriphoneLabels` converts phone labels into a context dependent form and also create a list of triphone models that appear in the training data.

```
createTriphoneLabels(outEdFile=mkTriEdFile, monoModels=spMonoModelList,
monoLabelFile=realignedMonoMLF)
```

[3]It is a good idea that you compare the graph of likelihood on training data and the graph of recognition accuracies on test data.

Make sure that a context dependent (triphone) label file, `labels/phone/xwrd_tri.mlf` has been created from `labels/phone/mono-aligned2.mlf` and a list of triphone models, `model_lists/xwrd_tri.list` has been created as well.

5.1.2 Step 1b: creating cross-word triphone models

Call `createTriphones` to create initial cross-word triphones by simply cloning monophone models in `models/R1/hmm9`.

```
createTriphones(inEdFile=mkTriEdFile, monoIter=9, triIter=0,
monoType=monoSingle, triType=xwrdTriSingle, monoModels=spMonoModelList)
```

The initial model will be stored under `models/R3/hmm0`.

Because the total file size of triphone models is much larger than that of monophone models, model files hear-after will be stored in binary format rather than ascii format for efficiency, and thus you cannot take a look at them directly.

5.1.3 Step 1c: training cross-word triphone models

Call `trainHMMs` to train the triphones up to hmm3 (or further).

```
trainHMMs(hmmIter=0, n=3, hmmType=xwrdTriSingle, modelFile=initialTriModelList,
labelFile=initialTriMLF, statsFlag=True, binaryFlag=True)
```

The cross-word triphone models created so far are only those triphones that appeared in the training set, and thus a lot of triphones might be missing.

5.2 Step 2: Single-Gaussian tied-state triphone models

5.2.1 Step 2a: clustering HMM states

`createTiedStateTriphones` will carry top-down clustering of HMM states starting from R3/hmm3

createTiedStateTriphones(inIter=3, outIter=0, inType=xwrdTriSingle, outType=tiedTriSingle)

which will produce a long log file which will include a line like this:

```
TB: Stats 24->1 [4.2%]  { 60309->2086 [3.5%] total }
```

which shows that a total of 60,309 HMM states were merged into 2,086 clustered.

`createTiedStateTriphones` creates those lists:

`model_lists/full_tri.list`	list of all possible triphone models
`model_lists/trees`	current questions defined and trees
`model_lists/treeg.list`	phyical models and associated logical models

See HTK Book section 3.3.2, 10.5, and 17.8 for details, and see the script file, `scripts/mk_tied_triphones` to understand how HHEd is called for clustering.

5.2.2 Step 2b: training tied-states triphone models

```
trainHMMs(hmmIter=0, n=3, hmmType=tiedTriSingle, modelFile=tiedTriModelList,
labelFile=initialTriMLF, statsFlag=True, binaryFlag=True)
```

5.2.3 Step 2c: evaluation

```
recogniseSpeech(hmmIter=3, hmmType=tiedTriSingle, modelFile=tiedTriModelList, tiedTriphoneFlag=True)
```

```
scoreWER(hmmIter=3, hmmType=tiedTriSingle)
```

If you are recognising with tied triphone states, then you must set `tiedTriphoneFlag=True`

5.3 Step 3: Mixup – multiple mixture triphone models

Much the same way for the monophone models, call copy models and `mixupGMM`

```
copyModels(hmmIter=3, oldType=tiedTriSingle, newType=tiedTriGMM)
```

```
mixupGMM(hmmIter=0, hmmType=tiedTriGMM, mixEdFile='edfiles/mixup2.hed',
modelFile=tiedTriModelList)
```

And then train in the usual way:

```
trainHMMs(hmmIter=1, n=3, hmmType=tiedTriGMM, modelFile=tiedTriModelList,
labelFile=initialTriMLF, statsFlag=True, binaryFlag=True)
```

For evaluation, calling `recogniseSpeech` and `scoreWER` can be called in a similar way to step 2c.

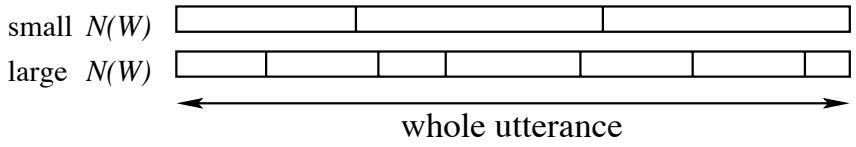
6 Implementation of decoding in HTK

ASR system uses $P(X|W)$ and $P(W)$ to find the best word sequence W.

- These estimated probabilities are different in reliability and dynamic range.
- More shorter words tend to have higher scores than those with fewer longer words in real implementation.

$$
\begin{aligned}
W^* &= \arg\max_W P(X|W)\,P(W) \\
&\Rightarrow \arg\max_W P(X|W)\,P(W)^{LMS}\,IP^{N(W)} \qquad \cdots \text{ modified formula} \\
&= \arg\max_W \log P(W|X) + LMS \log P(W) + N(W)\log IP
\end{aligned}
$$

LMS : language model weight (LM scaling factor)
IP : insertion penalty
$N(W)$: number of words in W



- Interpretation of *LMS*
  As $LMS \to 0$, $P(W)^{LMS} \to 1$,
  i.e. the smaller *LMS* becomes, the less important the LM is.

- Interpretation of *IP*
  Assuming a uniform LM (every word has an equal occurrence probability), $P(W)$ for fewer longer words (i.e. smaller $N(W)$) is greater than $P(W)$ for more shorter words (i.e. larger $N(W)$). *IP* is used to balance this.

$0 < IP \leq 1$ (i.e. $\log IP < 0$)	the smaller *IP* becomes, the more shorter words are penalised (i.e. fewer longer words are preferred)
$1 < IP$ (i.e. $\log IP > 0$)	the larger *IP* becomes, the more shorter words are preferred. (i.e. the more insertion errors.)

- *LMS* and *IP* are determined heuristically (on validation data). Larger *LMS* usually needs larger *IP*.

7 Assignment

You should complete both parts of the assignment (part 2 will be available on 28 February). Your coursework submission is complete only if you have done the following two submissions:

- submission of report (in PDF format)
- submission of your ASR system that gave the highest recognition accuracy

The submission deadline is Wednesday 27 March at 16:00.

Your coursework will be assessed mainly based on

- quality of experiments/investigation
- quality of report and discussions

Note that your coursework should be done solely by yourself, meaning you cannot use reports/documents, programs, files, or experimental results of someone else (except the programs/scripts provided in the course). However, you may have discussions with your colleagues. For plagiarism, please see:

`http://www.inf.ed.ac.uk/teaching/plagiarism.html`

7.1 Submission of report

1. Download a report template package from the course web page. There are two versions available, one for LaTeX, the other for Microsoft Word (Open Office). You can use either one.
2. As you will find in the package, your report submission should consist of the following two documents:
   (a) coversheet
   (b) report

   For the purpose of anonymous assessment of reports, do not write your name or matriculation number on your report, but write your "pseudonym" on the first page of the report. A pseudonym can be any word(s) or letter/digit sequence but your own name. On your coversheet, indicate your name, matriculation name, and pseudonym.
3. Submit your report and coversheet files with the `submit` command in DICE computing environment. The following shows an example of submitting two files, `report.pdf` and `coversheet.pdf`.

   ```
   % submit asr 1 report.pdf coversheet.pdf
   ```
4. You should receive an email of acknowledgement from the system as soon as your submission has been received successfully. Keep the message as an evidence of your coursework submission.

7.2 Submission of the best ASR system

This part can be minimal as long as you use *WorkDir* and you keep all the relevant files (i.e. all the models you trained, and other files needed to run the system) there. If it is not the case, please contact the course lecturer for advice.

Having all the relevant files stored under in *WorkDir*, please create a python script in *WorkDir*called `bestSystem.py`. It could be just a copy of your recognition script that gave the best result. Make sure that HMM model directory and HVite options are set properly in the script so that it gives the same result (accuracy) as the one for the best system described in your report.

7.3 Report writing

Your report should be prepared using report templates, which will be provided on the course web page shortly. Keep the length of your report between 5 and 8 pages including figures, tables, and references.

You should look into the following points and write a "scientific report". For higher marks, you need to give good discussions based on both theories and experiments, and you will also need to try the optional items.

1. Monophone models

- investigate how the three parameters (-t, -s, -p) of HVite influence recognition performance (i.e. accuracy and speed), and summarise the results using graphs, e.g. graphs of WER vs. pruning threshold, WER vs. speed, etc.
- find the optimal number of mixture components (optional)
- carry out experiments using different feature vectors, e.g. MFCC without delta features (optional)

2. Tied-state triphone models

   - investigate how the number of clusters influences recognition performance by varying the threshold value for "TB" command of HHEd (see `./scripts/mk_tied_triphones`). Draw a graph of recognition accuracy in terms of the total number of free model parameters (or the number of clusters).
   - seek the optimal configuration of parameters that gives highest recognition accuracy (optional).

3. MLLR speaker adaptation

   - investigate the relationship between the number of regression classes and recognition performance.
   - try adaptation experiments for different HMMs (e.g. different numbers of mixture components, triphone models) [optional]
   - investigate the relationship between the amount of training samples and recognition performance. [optional]
   - based on the experiments, discuss the limitation of this type of adaptation and what modification could be done. [optional]

The following are some technical instructions on writing reports.

- experimental results should be efficiently summarised using figures or tables (but not both if possible) - avoid using a separate graph/table for each experiment.
- figures/tables should be numbered and captioned.
- experimental conditions and methods should be shown clearly, using a table for example.(e.g. data sets used for training and evaluation, relevant parameters used for training and evaluation) Providing sufficient information is very important for scientific reports (see below) so that other people could redo the same experiments.
- show results even if there was no improvement. It is important to discuss/analyse why there was no improvement.
- for "scientific method" and "scientific report", please see

  http://en.wikipedia.org/wiki/Scientific_method
  http://www.unc.edu/depts/wcweb/handouts/lab_report_complete.html

Tips on experimental design

You will see that there are a large number of combinations of parameters which will affect training speed, recognition speed and recognition accuracy. Although it does not take much time for carrying out a single session of experiment with the default parameter set, it will take much longer for certain sets of parameters. In addition, it's not feasible to try all of the possible combinations of the parameters. Plan what experiments are needed and how they should be carried out.

To carry out experiments efficiently, it is a very good idea that you create script files which test various parameter combinations automatically.

You should start experiments as soon as possible in order not to miss the deadline.

Appendix: trainWSJCAM0.py

```
#!/usr/bin/python

from HTK_WSJCAM0 import *

# different phases of training / model complexity
monoSingle = 'R1'
monoGMM = 'R2'
xwrdTriSingle = 'R3'
xwrdTriGMM = 'R4'
tiedTriSingle = 'R5'
tiedTriGMM = 'R6'

# count of number of training iterations for each phase
monoSingleIter = 0
monoGMMIter = 0
xwrdTriSingleIter = 0
xwrdTriGMMIter = 0
tiedTriSingleIter = 0
tiedTriGMMIter = 0

# files that will be created by programmes in the training procedure
spMonoModelList = 'model_lists/mono-sp.list'
silEdFile = 'edfiles/sil.hed'
realignedMonoMLF = 'labels/phone/mono-aligned2.mlf'
mkTriEdFile = 'edfiles/mktri_xwrd.hed'



### Phase R1:  monophone models with single component Gaussians



### Flat start monophones - R1:hmmP
prepMonophones(hmmType=monoSingle, trainingID='hmmP', modelFile=initialMonoModelList)

### Initialise monophones using HInit and HRest - R1:hmmR
initMonophones(hmmType=monoSingle, prepID='hmmP', initID='hmmI', restID='hmmR',
modelFile=initialMonoModelList)

### Merge monophone models into a single file for efficiency - R1:hmm0
mergeMonophones(hmmType=monoSingle, inID='hmmR', outID='hmm0', modelFile=initialMonoModelList)

### Forward-backward training using HERest, 4 iterations - R1:hmm4
nIters = 4
trainHMMs(hmmIter=monoSingleIter, n=nIters, hmmType=monoSingle, modelFile=initialMonoModelList,
labelFile=initialPhoneTrainingMLF)
monoSingleIter += nIters

### Now create short pause models - R1:hmm5
createSPModel(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList,
edFile=silEdFile)
monoSingleIter += 1

### Forward-backward training with sp model using HERest, 2 iterations - R1:hmm7
nIters = 2
trainHMMs(hmmIter=monoSingleIter, n=nIters, hmmType=monoSingle, modelFile=spMonoModelList,
labelFile=initialPhoneTrainingMLF)
monoSingleIter += nIters
```

```
### Realign using the current models

alignMLF(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList,
edFile=mergeSpSilEd, realignedMLF=realignedMonoMLF)

### Forward-backward training with realigned MLF using HERest, 2 iterations - R1:hmm9
trainHMMs(hmmIter=monoSingleIter, n=nIters, hmmType=monoSingle, modelFile=spMonoModelList,
labelFile=realignedMonoMLF)
monoSingleIter += nIters

### Recognise and score the devtest set using the current models
recogniseSpeech(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList)
scoreWER(hmmIter=monoSingleIter, hmmType=monoSingle)

### Recognise and score using a small version of the devtest set using the current models
recogniseSpeech(hmmIter=monoSingleIter, hmmType=monoSingle, modelFile=spMonoModelList, smallFlag=True)
scoreWER(hmmIter=monoSingleIter, hmmType=monoSingle, smallFlag=True)


### Phase R2:  monophone models with GMMs - R2:hmm0
copyModels(hmmIter=monoSingleIter, oldType=monoSingle, newType=monoGMM) #R2:hmm0

### Mixup to 2-compenent GMMs - R2:hmm1
mixupGMM(hmmIter=monoGMMIter, hmmType=monoGMM, mixEdFile='edfiles/mixup2.hed',
modelFile=spMonoModelList)
monoGMMIter += 1

### Forward-backward training with 2-component Gaussians using HERest, 3 iterations - R1:hmm4
nIters = 3
trainHMMs(hmmIter=monoGMMIter, n=nIters, hmmType=monoGMM, modelFile=spMonoModelList,
labelFile=realignedMonoMLF)
monoGMMIter += nIters

### Recognise and score using a small version of the devtest set using the current models
recogniseSpeech(hmmIter=monoGMMIter, hmmType=monoGMM, modelFile=spMonoModelList, smallFlag=True)
scoreWER(hmmIter=monoGMMIter, hmmType=monoGMM, smallFlag=True)

### At this point you can continue to create models with more Gaussian components
### using mixupGMM() and trainHMMs()



### Phase R3:  cross-word triphone models

### create triphone labels and triphone model list
createTriphoneLabels(outEdFile=mkTriEdFile, monoModels=spMonoModelList,
monoLabelFile=realignedMonoMLF)

### create single-Gaussian cross-word triphones by cloning single-Gaussian monophone models
createTriphones(inEdFile=mkTriEdFile, monoIter=monoSingleIter, triIter=xwrdTriSingleIter,
monoType=monoSingle, triType=xwrdTriSingle, monoModels=spMonoModelList)

### train single Gaussian cross-word triphones - R3:hmm3
nIters=3
trainHMMs(hmmIter=xwrdTriSingleIter, n=nIters, hmmType=xwrdTriSingle, modelFile=initialTriModelList,
labelFile=initialTriMLF, statsFlag=True, binaryFlag=True)
xwrdTriSingleIter += nIters
```

```
### Phase R5:  tied cross-word triphone models - R5:hmm0
createTiedStateTriphones(inIter=xwrdTriSingleIter, outIter=tiedTriSingleIter, inType=xwrdTriSingle,
outType=tiedTriSingle)

#  train tied cross-word triphone models - R5:hmm3
nIters=3
trainHMMs(hmmIter=tiedTriSingleIter, n=nIters, hmmType=tiedTriSingle, modelFile=tiedTriModelList,
labelFile=initialTriMLF, statsFlag=True, binaryFlag=True)
tiedTriSingleIter += nIters

### Phase R6:  tied cross-word triphone models with GMMs - R6:hmm0
copyModels(hmmIter=tiedTriSingleIter, oldType=tiedTriSingle, newType=tiedTriGMM)

### Mixup to 2-compenent GMMs - R6:hmm1
mixupGMM(hmmIter=tiedTriGMMIter, hmmType=tiedTriGMM, mixEdFile='edfiles/mixup2.hed',
modelFile=tiedTriModelList)
tiedTriGMMIter += 1

#  train tied cross-word triphone models with - R6:hmm4
nIters=3
trainHMMs(hmmIter=tiedTriGMMIter, n=nIters, hmmType=tiedTriGMM, modelFile=tiedTriModelList,
labelFile=initialTriMLF, statsFlag=True, binaryFlag=True)
tiedTriGMMIter += nIters

recogniseSpeech(hmmIter=tiedTriGMMIter, hmmType=tiedTriGMM, modelFile=tiedTriModelList,
tiedTriphoneFlag=True)
scoreWER(hmmIter=tiedTriGMMIter, hmmType=tiedTriGMM)
```